Los Alamos
NATIONAL LABORATORY
EST. 1943

LA-UR-20-20085
Approved for public release; distribution is unlimited.

Title: Simtools Project Overview

Author(s): Schmidt, Joseph H.

Intended for: Presentation for discussion with colleagues at LLNL.

Issued: 2020-01-07

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

SimTools Project Overview

Joe Schmidt

January 6, 2020

Operated by Triad National Security, LLC for the U.S. Department of Energy's NNSA

The SimTools project members are Travis Drayna, Laura Lang, Rachel Ertl, Louie Long, Brian Jean, Henry Stam and Wally Atterberry.

SimTools Project Objectives

- End-to-end simulation data management
  - Enforce a consistent simulations process to help ensure differences in results are real and not the product of different data treatments.
  - Provide traceability of the simulation data and repeatability of the simulation process.
  - Provide controlled variation of input data and analysis.
- Provide ”force multipliers” for analysts.
  - Enable concise high-level problem descriptions.
  - Run multiple physics codes from a single problem description.
  - Automate input processing, job control and results analysis.
- **Rapidly answer questions requiring hundreds or thousands of simulations.**

We accomplish this by developing a *language* to describe the process for managing the simulations required for an analysis.

We have guidelines for developing this *language*.

Good Software Practices for Scientific Analysis

- **Once and only once** – Define source data and/or relationships only once.
- **Maintain relationship metadata** – Explictly define relationships among input data.
- **Human-readable input** – Easily verify source data, relationships and processing. Brevity.
- **Do not store generated data** – Script driven operation

$$\text{source data} + \text{script} = \text{problem setup}$$

- **Distinguish between data and modeling choices** – Changing modeling choices should be straight forward.
- **Open architecure** – New capabilities can be added without the assistance of the development team.

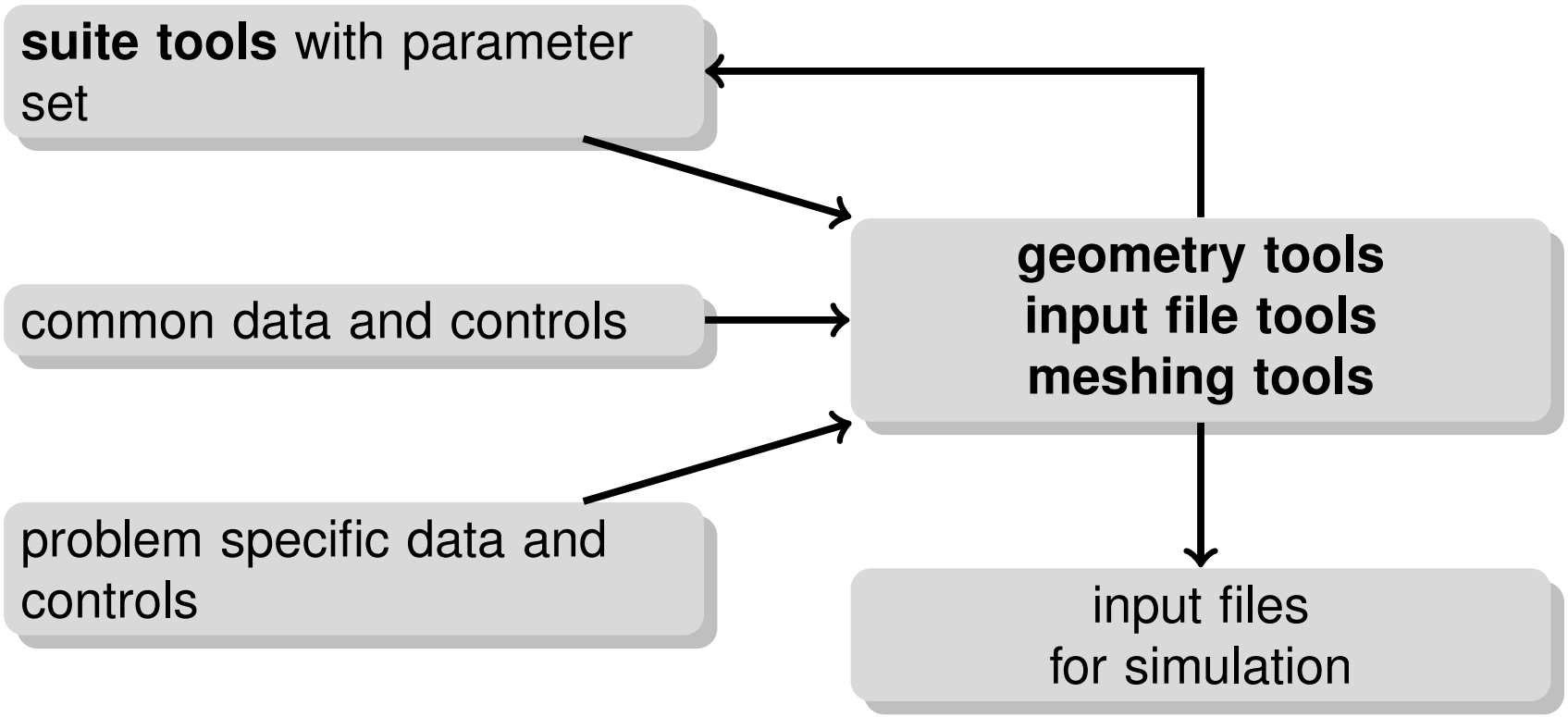
Problem

- An analysis can require hundreds of simulations.
- Each simulation requires lengthy input files. For a 3D problem with 1000 vertices in each direction, we require the coordinates of 10^9 vertices. At 24 bytes per vertex, that is 24Gb.
- The coordinate values are the wrong language for thinking about the problem.
- We need a language that describes the pattern of the mesh coordinates concisely to promote understanding, repeatability and traceability.
- The same argument applies to material properties, control of the solution algorithm, . . . — to every aspect of the problem. The language that we use to describe the simulation influences how we think about the problem.
- The role of the SimTools project is to develop a language for problem solving that improves the efficiency and capability of our users.

Solution

- We start with Python, a complete and capable language.
- We add modules that contain nouns (objects) and verbs (methods) that pertain to physics simulations – essentially a glossary of available language.
- Users write scripts using the Python syntax with the SimTools language to describe their modeling process.
- The SimTools project looks for verbose and complicated constructs in the users work as opportunities to add to the language in a way that improves traceability and repeatability.
- The biggest trap is making easy things easier and there by making hard things impractical. Our goal is to make the hard things practical.

Outline


suite tools with parameter set
common data and controls
problem specific data and controls
geometry tools
input file tools
meshing tools
input files for simulation

We can break down our tools for managing simulations into four categories, tools that

- Create and manage geometry for experiments (ingen.gwiz, ingen.gwiz.solid, etch, osito, calico),
- Support the setup of suites of calculations (ingen.suite, ingen.csv),
- Construct input files in a robust manner (ingen.tengwar)
- And build boundary fitted meshes (ingen.altair, charybdis).

Geometry

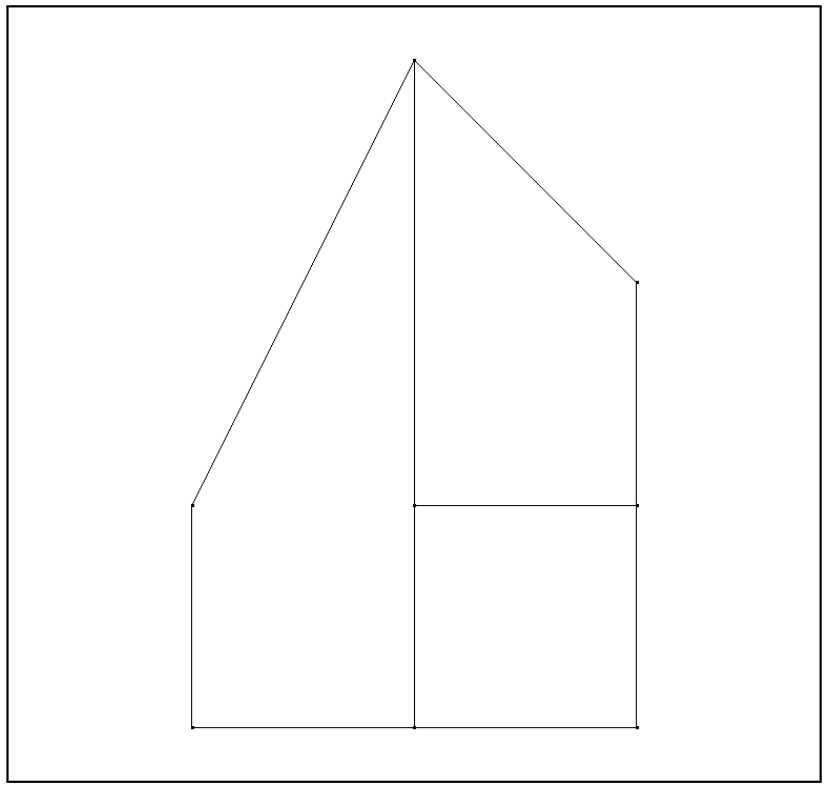
1. Contour file format
2. Reading contour files
3. Creating contours in Python
4. Painting solid geometry and parts
5. Multiple, related geometries
6. Contours and parts

1. The contour file format is the best format we have for capturing pedigree – information about where the data comes from in an easy to check form.
2. Once you have a directory of contour files, reading them with ingen.gwiz is simple.
3. Sometimes the geometry is easier to create by an algorithmic process and ingen.gwiz has commands to do that.
4. If one does not intend to make a mesh conform to geometry, then the geometry can be constructed as volumes using ingen.gwiz.solid for various primitive geometric objects and their unions, intersections and differences. Calico calculates fractions of materials in zones from the volumes. Parts are objects that combine the geometry and material information. We can create parts using the solids combined with material objects.
5. For related experiments, ingen.lcars reads from a directory structure in a manner that allows the simulations to share geometry.
6. We can create parts using contours combined with material objects.

Geometry – Contours – Contour File Format

The contour file format defines a contour as a sequence of directed curvilinear pieces (*r*-*z*, *r*-θ, circle, ellipse, line and mirror).

```
piece =
  line(start=(1.0cm, 0.0cm),
  end=(3.0cm, 1.0cm)
```



The basic component of a contour in the contour file format is the piece. We can define a piece as a line, an arc, a table or other primitive. A contour consists of a sequence of these pieces linked together. Using a sequence of pieces lets us describe complicated contours.

Geometry – Contours – from .contour File to Python

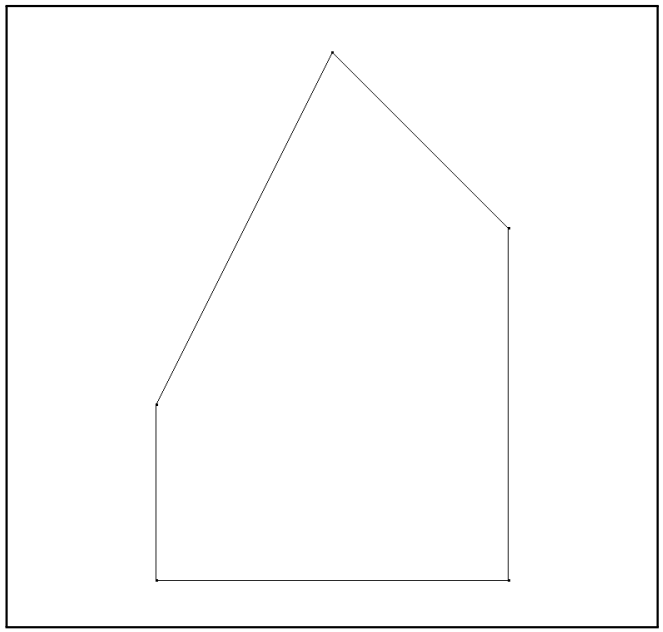
Given a directory named *geom* with a collection of *.contour* files, the Python to read the files is:

```
from ingen import init, gwiz
init(globals(), None)
gwiz.loadDirectory('geom',
     [("*", gwiz.noRule(), gwiz.nDistribRule(5))],
     cntr)
```

- Our collection of tools is made available in a collection of Python modules which can be imported. In the remaining examples, the import statements will not be shown in the interest of brevity.
- We provide an *init* method to create a conventional set of namespaces. Users can create additional namespaces, but at this time our graphical interface uses the conventional namespaces for visual display.
- Finally, a single command reads the contour files and assigns a resolution to the contours. We have found it useful to be able to specify distinct resolutions for tabular pieces and analytic pieces. The noRule shown uses the points in the tabular pieces without additional spline fit points). The nDistribRule shown uses 5 points for each analytic piece.
- Typed namespaces are containers for **typed** objects. As we name objects, there tend to be many objects that ought to get the same name – namespaces help us avoid name collisions. They provide a mechanics that groups objects within the language in a manner akin to prefixed variable names: cntrC1. The *init* method defines several default namespaces: *pnt*, *cntr*, *reg*, *seg*, *blk*, *blk1*, *mat*, *sid*, *per* and *prt*.

Geometry – Contours – Contours Constructed in Python Directly

```
p2 = gwiz.rzPoint((1.0, 0.0))
p3 = gwiz.rzPoint((3.0, 1.0))
cntr.c1 = gwiz.line(p2, p3)
```



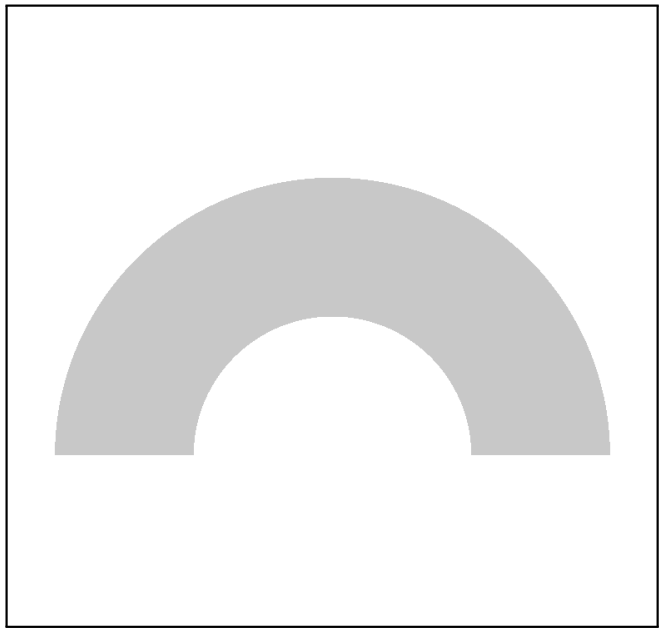
For contours best described by a process, we can create and modify contours with the *gwiz* library.

In this example, we define points with *gwiz.rzPoint* and connect them to make a line with *gwiz.line*.

Geometry – Constructive Solid Geometry – CSG

```
generic = materials.generic()
Cu = generic()
reg.s1 = solid.sphere((0.0, 0.0, 0.0), 2.0)
reg.s2 = solid.sphere((0.0, 0.0, 0.0), 1.0)
reg.diff = reg.s1-reg.s2
```

```
myManifest = parts.manifest(reg, prt)
myManifest.diff(Cu(density=9.0))
```



For geometry that we will not mesh, we can specify it via constructive solid geometry using the *gwiz.solid* library. We support a variety of primitive shapes including spheres, boxes, spun tabulars, ellipsoids, cones, ... Further we can construct more complicated volumes via set arithmetic with unions, intersections and differences.
In this example, we create two primitive spheres with *gwiz.solid.sphere* and subtract them (a set difference operation) to make a hollow sphere.

After making the volume, we convert it into a part by combining the geometry with material information.

Geometry – Managing Multiple, Related Geometries

```
myLcars/:
./  ../  dirs/ exp1/  .lcars

myLcars/dirs:
./  ../  exp1@ exp2@  exp3@

myLcars/exp1:
./  c01.cntr@   c02.cntr@   c03.cntr@   c0[45].cntr@   exp2/
../ c01.contour c02.contour c03.contour c0[45].contour exp3/

myLcars/exp1/exp2:
./  c05.cntr@   c06.cntr@   c07.cntr@
../ c05.contour c06.contour c07.contour

myLcars/exp1/exp3:
./  c04.cntr@   c05.cntr@   c0[6789].cntr@   c10.cntr@
../ c04.contour c05.contour c0[6789].contour c10.contour
```

- We use a directory structure as a tree data structure for storing multiple related geometries.
- LCARS reads the information from the tree. Each LCARS directory contains a *.lcars* file with the LCARS version, a *dirs* directory that acts as a table of contents with file pointers to the lowest directory for each experiments geometry, and a collection of directories that store the geometry.
- Geometry for experiments in subdirectories is inherited from the geometry in the parent directories. In the example shown, *exp1* is a top level experiment and its geometry consists solely of the contours in its directory, *c01* through *c05*. *exp2* inherits *c01* through *c04* from *exp1*, over writes *c05* with the definition in its directory and adds contours *c06* and *c07*.
- To allow for a distinction to be made between types and formats, the type is given by a softlink in the directory *c01.cntr* indicates that *c01* is a contour while *c01.contour* indicates that it is a file in the .contour format as opposed to a contour in a .rz format.

Geometry – Managing Multiple, Related Geometries

Loading exp2 makes the contours *c01*, *c02*, *c03*, *c04*, *c05*, *c06* and *c07* available, where *c05* is the contour from *myLcars/exp1/exp2* directory.

```
lib = lcars.open('myLcars')
experimentDictionary = lib.get('exp1', gwiz.Contour, None)
lcars.load(experimentDictionary,
           (cntr, gwiz.noRule(), gwiz.nDistribRule(5)))
```

Geometry – Contours – Parts

Going from contours to sides introduces topology:

```
parts.contours2Sides(cntr, sid)
outsideChain = parts.chain(sid.c03,sid.c01,sid.c02,sid.c04)
per.main = parts.stack(outsideChain).fill()
per.house1, per.house2 = per.main.cut(sid.c06)
del per.main
```

Adding materials to perimeters creates parts:

```
myManifest = parts.manifest(per, prt)
myManifest.house1(Cu(density=9.0))
myManifest.house2(Cu())
```

- Sides are the beginning of topology. A side is a contour that knows the areas on either side of it.
- A sequence of sides is a chain.
- A closed chain is a perimeter.
- The advantage of the perimeter over the contour is that it identifies a region in space – the geometric portion of a part. The advantage of a side is that it can identify the perimeters it separates. Having a notion of topology allows for geometric operations on parts.
- One advantage of namespaces is the automatic promotion of objects – in this example from perimeters to parts.
- Adding the material information to the geometry makes a part.

Suites

Simulation control for the many simulations in an analysis.

1. CSV tables
2. Setup directory
3. Creating a suite of simulations

1. To make a suite of simulations, we need to parameterize the differences in a simulation. We keep those differences in tables in CSV, comma separated values, form. The CSV table can be stored in Python form or read from a file.
2. We also need a set of files for the simulations where the parameters are going to appear with substitutions of parameters making each simulation distinct.
3. The Python to instantiate the suite is brief.

Suites – Managing Multiple Simulations – CSV Tables

```
table = \
'''
key,   list, var1, var2, var3
k1 , myRuns,    1,    2,    3
k2 , myRuns,    2,    4,    6
k3 , myRuns,    3,    6,    9
'''
```

This example shows the Python form of the CSV table.

Suites – Managing Multiple Simulations – Setup Directory

- setup/file1
- setup/file2

file1:

```
The value of the first variable is {var1}.
The value of the second variable is {var2}.
The value of the third variable is {var3}.
```

We have two files for each simulation directory. The substitution patterns are the column names from the CSV table in curly braces.

Suites – Managing Multiple Simulations

```
myDb = csv.db()
myDb.parse(string = table)
mySuite = suite.suite(myDb, './runs')
myRuns = mySuite.pick('myRuns')
for run in myRuns:
  run.setup(src=["setup"], tr=['*'])
```

- We begin by creating a database object to hold the parameters for the suite.
- We read the list of parameters sets from the table shown earlier.
- In a CSV, we allow multiple tables for constructing complicated suites. We use list columns in the CSV tables as tags for the rows. We select the rows for a set of runs for a suite using a tag – in this example, the tag is *myRuns*.
- Finally, we can loop over the runs and call setup which creates the directory for each simulation in the suite. We tell it the source directory to look for files. We can copy files from the source directory. We can softlink to files from the source directory. We can translate files from the source directory – that is we read the file and substitute for the expressions in curly braces. In this example we are only translating the files.

Suites – Managing Multiple Simulations

```
(:-> ls -CFR runs
runs:
k1/  k2/  k3/

runs/k1:
file1  file2

runs/k2:
file1  file2

runs/k3:
file1  file2
```

The contents of *runs/k1/file1* is

```
The value of the first variable is 1.
The value of the second variable is 2.
The value of the third variable is 3.
```

The result for this example is a runs directory with prepared for simulations *k1*, *k2* and *k3*. Each directory contains the files *file1* and *file2*. The contents of *file1* with the curly braced parameters replaced for the *k1* is shown.

Input Files

1. Flag
2. Rage

We manage the contruction of input files because of

- the volume of information in input files,
- the duplicate information within an input file,
- the duplicate information between input files,
- the duplicate information between input files for different codes,
- and the connections between the geometry and mesh and the material information.

The duplicate information and the connnections need to be placed in the input file in a robust and consistent manner. The problem of input file construction demands automation. Even more so when making suites of calculations.

Input Files – Separate Input File Format from Content – Flag

```
d=flag.Deck()
i=flag.Instance('/stuff',a=1)
d.addb(i)
d.addb(i.child('foo',b=2))
print(d)
```

```
mk /stuff
  a = 1
mk /stuff/foo
  b = 2
```

- We separate the format of the information in the input file from the values.
- We store the values in a tree structure (deck, instance and block objects).
- We use specializations of the objects (flag.) that know the syntax to use to write the values.

The tree structures are something that can be manipulated to maintain consistent information. When we print the tree structures, we get the input file in the correct syntax.

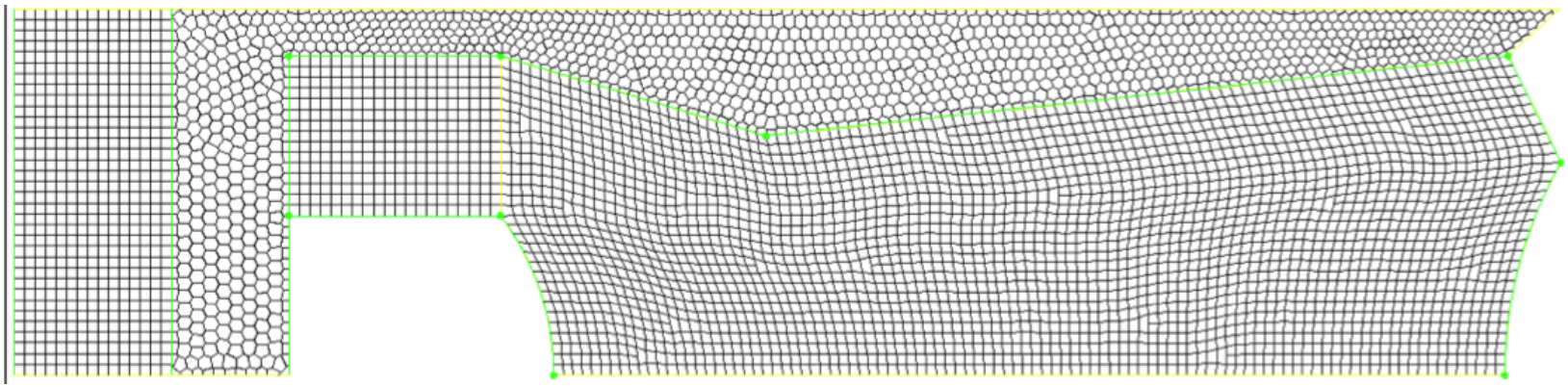
Input Files – Separate Input File Format from Content – Rage

```
d=rage.Deck()
d.add(a=1,b=2)
d.addi(5,c=3)
ib=rage.ifBlock('foo .eq. True',d=4)
d.addb(ib)
print(d)
```

```
a = 1
b = 2
c(5) = 3
if (foo .eq. True) then
  d = 4
endif
```

Meshing

1. Single block
2. Multi-block
3. Partial face match
4. Dendrites
5. Parts
6. Feature painting



Our meshing library is

1. block structured which makes it practical to align lines in the mesh with expected symmetries in the calculation,
2. multi-block which makes each block a large ”element” within the overall mesh – provides unstructured mesh flexibility,
3. partial face matching which reduces the number of blocks into which the problem is split,
4. dentritic which allows direction control with size control,
5. part aware so that topology information from the geometry can be carried across to the topology of the mesh
6. and makes volume fractions for parts without a conforming mesh.

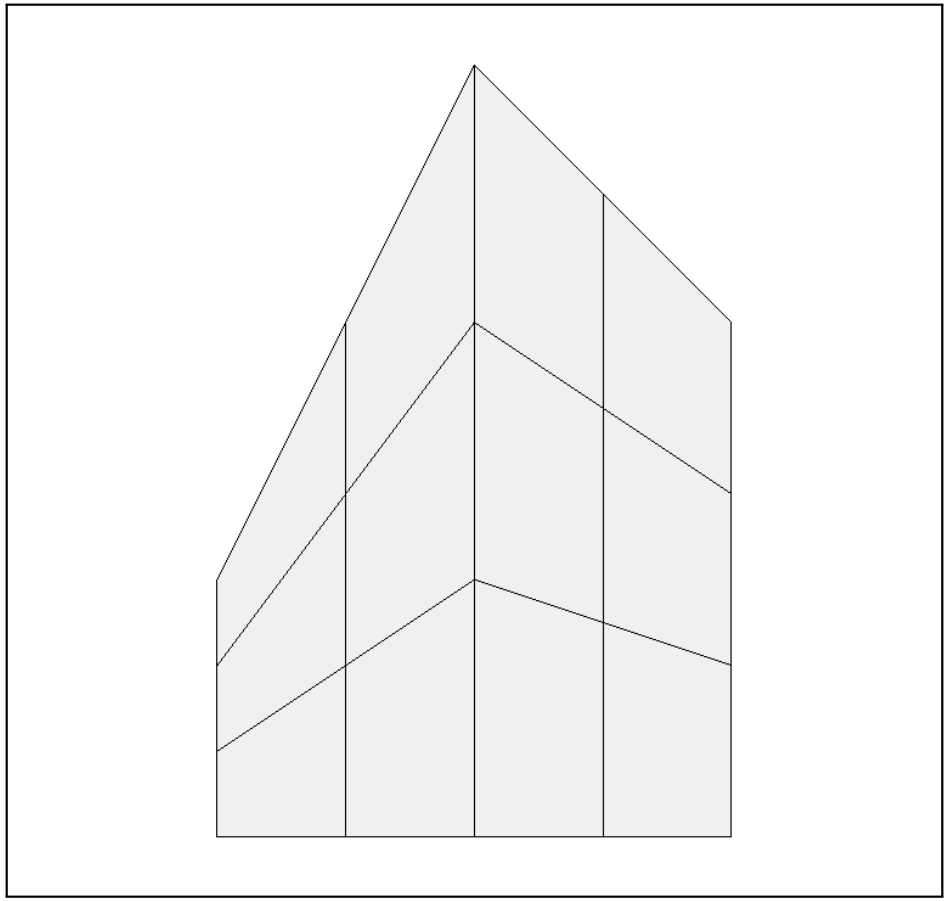
In addition, we can spin the 2D meshes into 3D meshes.

Meshing – Single Block Mesh

```
altair.contours2Segments(cntr, seg)
seg.c01.equalArcDistrib(npts=3)
seg.c02.equalArcDistrib(npts=3)
seg.c03.equalArcDistrib(npts=4)
seg.c04.equalArcDistrib(npts=len(seg.c03))
seg.c05.equalArcDistrib(npts=len(seg.c01+seg.c02))
blk.exmpl1 = altair.block4(iMin=seg.c03, iMax=seg.c04,
                           jMin=seg.c05, jMax=seg.c01+seg.c02,
                           material=mat.Cu)
altair.finalize()
```

In this example, we promote the contours to segments using *contours2Segments* and then specify the number of points to discretize the segments, making the segments 1D meshes. Note that we can enforce consistency by specifying the number of points on a segment in terms of the number of points on another segment as shown with segment *c04*. When we have discretized the segments bounding a block, we use a block call to construct the block based on the surrounding segments.

Meshing – Single Block Mesh

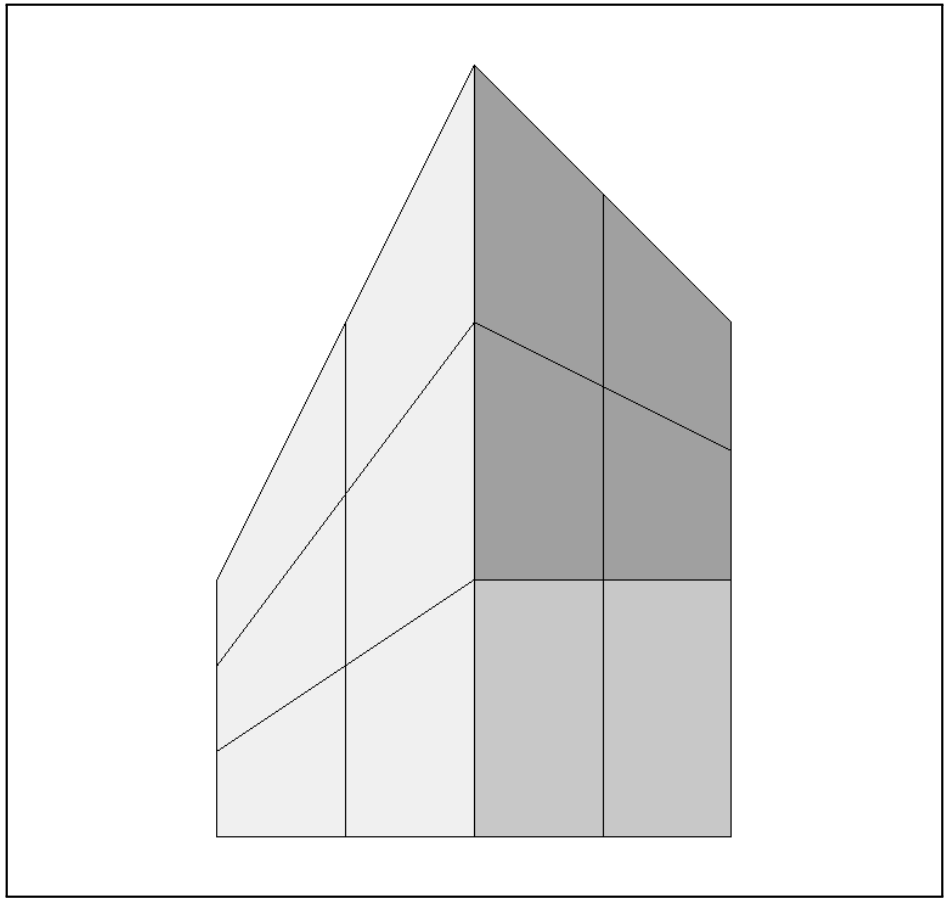


Meshing – Multi-Block, Partial Face Match

```
altair.contours2Segments(cntr, seg)
seg.c01.equalArcDistrib(npts=3)
.
.
.
seg.c10.equalArcDistrib(npts=len(seg.c09))
blk.exmpl1 = altair.block4(iMin=seg.c03, iMax=seg.c06+seg.c09,
                           jMin=seg.c05, jMax=seg.c01,
                           material=mat.Cu)
blk.exmpl2 = altair.block4(iMin=seg.c06, iMax=seg.c04,
                           jMin=seg.c07, jMax=seg.c08,
                           material=mat.Cu)
blk.exmpl3 = altair.block4(iMin=seg.c09, iMax=seg.c10,
                           jMin=seg.c08, jMax=seg.c02,
                           material=mat.Cu)
altair.finalize()
```

- Using the segments in multiple blocks connects the blocks.
- Adding segments together make longer segments which is our mechanism for partial face matching.
- Using the individual segments in one set of blocks (*c06* in the second block and *c09* in the third block) and using the sum in another (*seg.c06+seg.c09* in the first block) enforces the connectivity for partial face matching.

Meshing – Partial Face Match

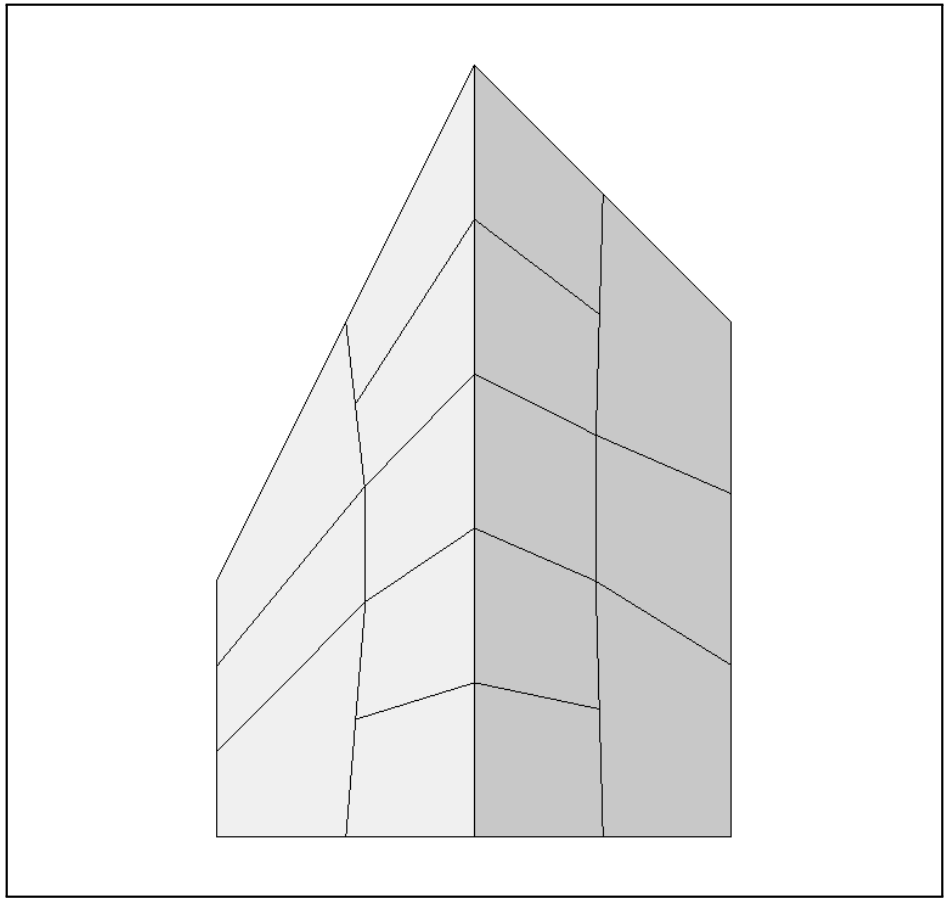


Meshing – Dendritic Meshing

```
altair.contours2Segments(cntr, seg)
seg.c01.equalArcDistrib(npts=3)
.
.
.
seg.c07.equalArcDistrib(npts=len(seg.c02))
blk.exmpl1 = altair.block4(iMin=seg.c03, iMax=seg.c06,
                           jMin=seg.c05, jMax=seg.c01,
                           material=mat.Cu, feather=altair.fthr())
blk.exmpl2 = altair.block4(iMin=seg.c06, iMax=seg.c04,
                           jMin=seg.c07, jMax=seg.c02,
                           material=mat.Cu, feather=altair.fthr())
altair.finalize()
```

Choosing different resolutions for opposite faces of a block (iMin/iMax or jMin/jMax) allows greater control of resolutions in a problem. We then adjust the resolution across the block by feathering out the change in a sequence of dendrites where mesh lines terminate.

Meshing – Dendritic Meshing



The key to the algorithm is to pick the order that the edges are removed, calculate the number of edges to remove for each column and do so in order.

Meshing – Parts Based Meshing

1. Start with parts.
2. Promote the materials from physical material objects to simulation material objects.
3. Promote parts to quilts.
4. Create the segments.
5. Distribute points on the segments.
6. Create the blocks.

- We define a topology for the geometry so that we can consistently manipulate the geometry.
- We define a topology for the mesh with the boundary meshes.
- In parts based meshing, we carry over the topology from the geometry to the mesh to eliminate needlessly specifying the topology twice.

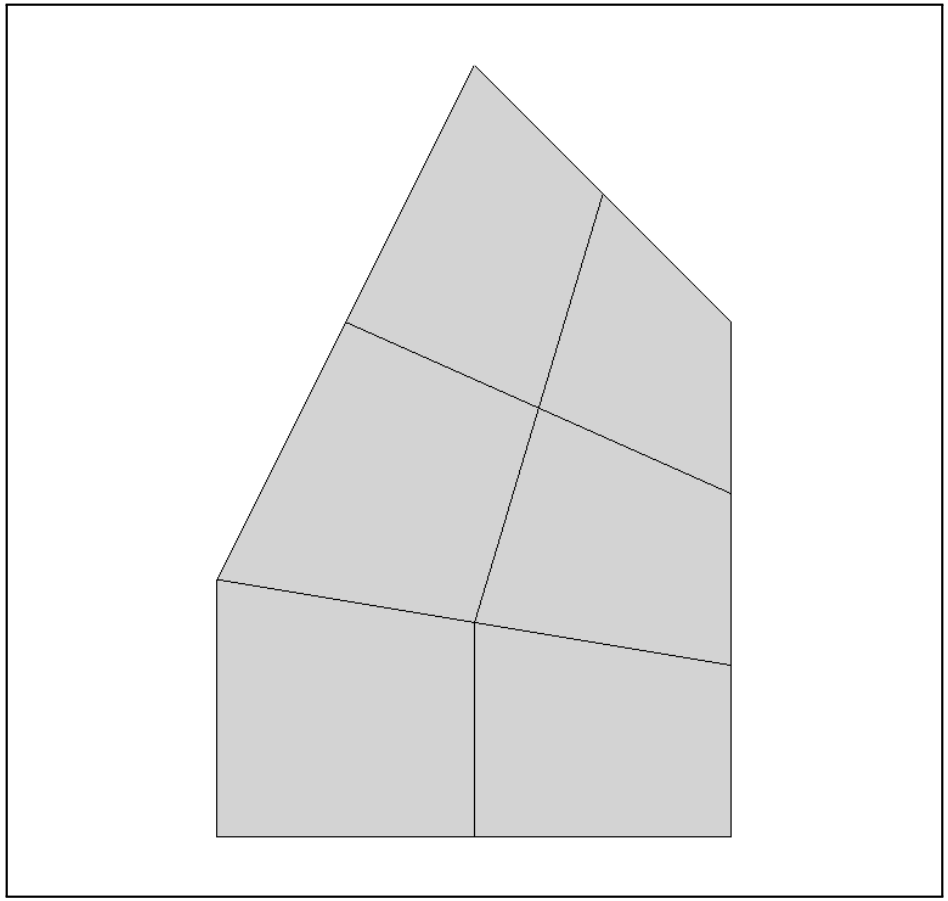
Meshing – Parts Based Meshing

```
parts.simulateMaterials(prt, mat)
parts.parts2Quilts(prt, qlt)
parts.sew(qlt)
parts.makeSegments(qlt)
myFrame = qlt.house.getFrame()
myFrame.iMin.equalArcDistrib(npts=3)
myFrame.jMin.equalArcDistrib(npts=4)
myFrame.copy()
parts.quilts2Blocks(qlt, blk)
altair.finalize()
```

In this example:

- We convert the physical materials on the parts to simulation materials.
- We then convert the parts to a quilts – a mesh block that has not been filled with zones.
- We get the frame – which holds the segments for the part – and set the resolutions.
- Finally, we call quilts2Blocks to fill in the zones.

Meshing – Parts Based Meshing



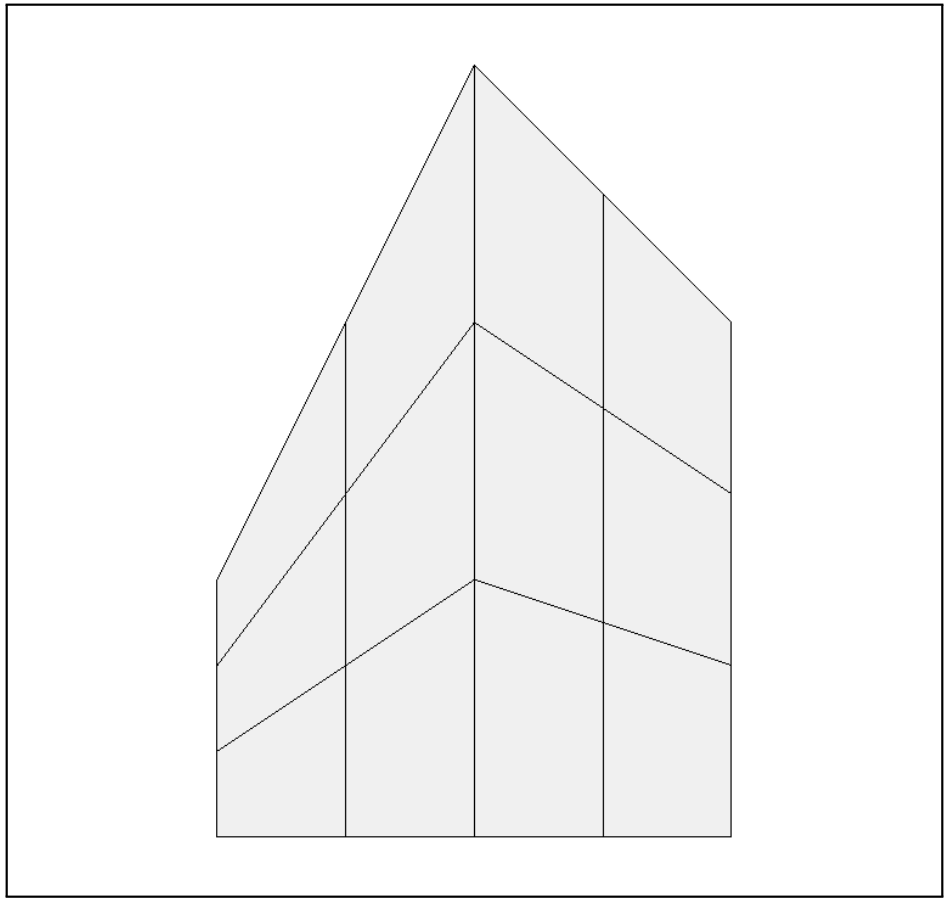
This instance indicates the principal problem with the automatic transfer of the topology. The topology of the geometry does not have a notion of an i direction or a j direction. The automatic transfer may not give the desired alignment. In the example, this problem is further exacerbated by the fact that the initial geometry has 5 sides rather than 4.

Meshing – Parts Based Meshing with Corner Control

```
sel.minusR = gwiz.extreme('-r')
...
pnt.leftBottom = sel.minusR(cntr.c03)
...
sel.nearLeftBottom = gwiz.nearest(pnt.leftBottom)
...
qlt.house.corner(sel.nearRightTop)
qlt.house.corner(sel.nearRightBottom)
qlt.house.corner(sel.nearLeftTop)
qlt.house.corner(sel.nearLeftBottom)
...
```

- We can manually intervene with control points.
- We call the control points corners.
- There are four corners per block.
- We first select the points using extremes of the contours, then locate the nearest points of the perimeter. After adjusting the resolutions, we get the same single block mesh.
- This is a place where our language needs work.

Meshing – Parts Based Meshing with Corner Control



Conclusion

- We develop the contour file format, Python contour methods, CSG methods and LCARS to help users manage the geometry of their simulations.
- We make tools to parse CSV tables and produce setups for suites of simulations.
- We provide tools to separate the format from the values in an input file so that users can consistently construct input files.
- We have written a multi-block, partial face matching, dendritic, part aware meshing library with which users can make boundary fitted meshes.